CAN 总线

CAN 是控制器局域网络(Controller Area Network，CAN)的简称，由德国 BOSCH 公司开发，并最终成为国际标准(ISO 11898-1)。CAN 总线主要应用于工业控制和汽车电子领域，是国际上应用最广泛的现场总线之一。

1 CAN 总线简介

CAN 总线是一种串行通信协议，能有效地支持具有很高安全等级的分布实时控制。 CAN 总线的应用范围很广，从高速的网络到低价位的多路接线都可以使用 CAN。在汽车电子行业里，使用 CAN 连接发动机的控制单元、传感器、防刹车系统等，传输速度可达 1 Mbps。

与前面介绍的一般通信总线相比，CAN 总线的数据通信具有突出的可靠性、实时性和灵活性，在汽车领域的应用最为广泛，世界上一些著名的汽车制造厂商 都采用 CAN 总线来实现汽车内部控制系统与各检测和执行机构之间的数据通信。目前，CAN 总线的应用范围已不仅仅局限于汽车行业，而且已经在自动控制、航 空航天、航海、过程工业、机械工业、纺织机械、农用机械、机器人、数控机床、医疗器械及传感器等领域中得到了广泛应用。

CAN 总线规范从最初的 CAN 1.2 规范(标准格式)发展为兼容 CAN 1.2 规范的 CAN 2.0 规范(CAN 2.0A 为标准格式，CAN 2.0B 为扩展格式)，目前应用的 CAN 器件大多符合 CAN 2.0 规范。

2 CAN 总线的工作原理

当 CAN 总线上的节点发送数据时，以报文形式广播给网络中的所有节点，总线上的所有节点都不使用节点地址等系统配置信息，只根据每组报文开头的 11 位标识符(CAN 2.0A 规范)解释数据的含义来决定是否接收。这种数据收发方式称为面向内容的编址方案。

当某个节点要向其他节点发送数据时，这个节点的处理器将要发送的数据和自己的标识符传送给该节点的 CAN 总线接口控制器，并处于准备状态；当收到总 线分配时，转为发送报文状态。数据根据协议组织成一定的报文格式后发出，此时网络上的其他节点处于接收状态。处于接收状态的每个节点对接收到的报文进行检 测，判断这些报文是否是发给自己的以确定是否接收。

由于 CAN 总线是一种面向内容的编址方案，因此很容易建立高水准的控制系统并灵活地进行配置我们可以很容易地在 CAN 总线上加进一些新节点而无须在硬件或软件上进行修改。

当提供的新节点是纯数据接收设备时，数据传输协议不要求独立的部分有物理目的地址。此时允许分布过程同步化，也就是说，当总线上的控制器需要测量数据时，数据可由总线上直接获得，而无需每个控制器都有自己独立的传感器。

3 CAN 总线的工作特点

CAN 总线的有以下三方面特点：

可以多主方式工作，网络上的任意节点均可以在任意时刻主动地向网络上的其他节点发送信息，而不分主从，通信方式灵活。

网络上的节点(信息)可分成不同的优先级，可以满足不同的实时要求。

采用非破坏性位仲裁总线结构机制，当两个节点同时向网络上传送信息时，优先级低的节点主动停止数据发送，而优先级高的节点可不受影响地继续传输数据。

4 CAN 总线协议的层次结构

与前面介绍的简单总线逻辑不同，CAN 是一种复杂逻辑的总线结构。从层次上可以将 CAN 总线划分为三个不同层次:

图 4-19　CAN 总线协议的三层结构

(1) 物理层

在物理层中定义实际信号的传输方法，包括位的编码和解码、位的定时和同步等内容，作用是定义不同节点之间根据电气属性如何进行位的实际传输。

在物理连接上，CAN 总线结构提供两个引脚--CANH 和 CANL，总线通过 CANH 和 CANL 之间的差分电压完成信号的位传输。

在不同系统中，CAN 总线的位速率不同；在系统中，CAN 总线的位速率是唯一的，并且是固定的，这需要对总线中的每个节点配置统一的参数。

(2) 传输层

传输层是 CAN 总线协议的核心。传输层负责把接收到的报文提供给对象层，以及接收来自对象层的报文。传输层负责位的定时及同步、报文分帧、仲裁、应答、错误检测和标定、故障界定。

(3) 对象层

在对象层中可以为远程数据请求以及数据传输提供服务，确定由实际要使用的传输层接收哪一个报文，并且为恢复管理和过载通知提供手段。

5 CAN 总线的报文结构

CAN 总线上的报文传输由以下 4 个不同的帧类型表示和控制。

(1) 数据帧

数据帧携带数据从发送器至接收器。总线上传输的大多是这种帧。从标识符长度上，又可以把数据帧分为标准帧(11 位标识符)和扩展帧(29 位标识符)。

数据帧由 7 个不同的位场组成：帧起始、仲裁场、控制场、数据场、CRC 场、应答场、帧结束。其中，数据场的长度为 0~8 个字节。标识符位于仲裁场中，报文接收节点通过标识符进行报文滤波。帧结构如图所示。

图 4-20　数据帧的结构

(2) 远程帧

由总线上的节点发出，用于请求其他节点发送具有同一标识符的数据帧。当某个节点需要数据时，可以发送远程帧请求另一节点发送相应数据帧。与数据帧相比，远程帧没有数据场，结构如图所示。

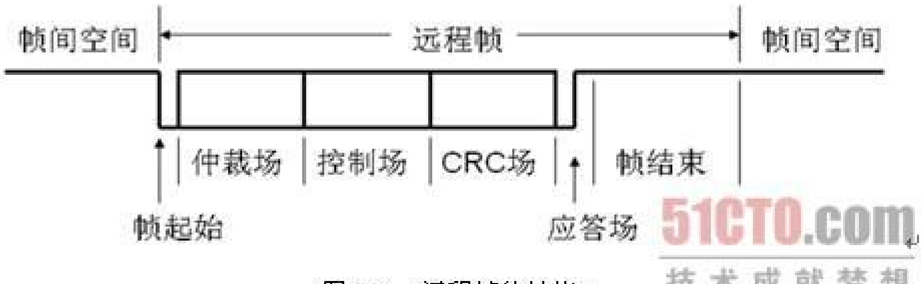

图 4-21　远程帧的结构

(3) 错误帧

任何单元，一旦检测到总线错误就发出错误帧。错误帧由两个不同的场组成，第一个场是由不同站提供的错误标志的叠加(错误标志)，第二个场是错误界定符。帧结构如图所示。

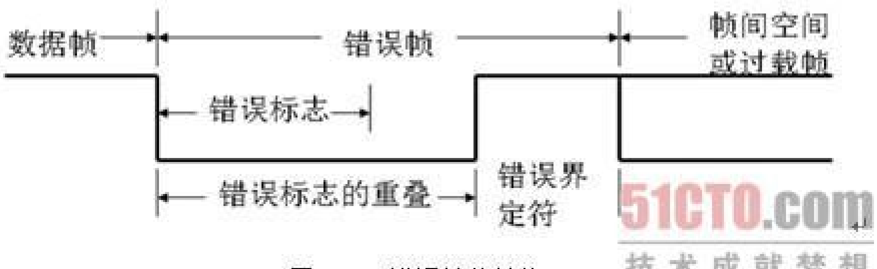

图 4-22　错误帧的结构

4. 过载帧

过载帧用于在先行的和后续的数据帧(或远程帧)之间提供附加延时。过载帧包括两个场：过载标志和过载界定符。帧结构如图所示。

图 4-23　过载帧的结构

6 CAN 总线配置

在 Linux 系统中，CAN 总线接口设备作为网络设备被系统进行统一管理。在控制台下， CAN 总线的配置和以太网的配置使用相同的命令。

在控制台上输入命令：

```
ifconfig –a
```

可以得到以下结果：

```
can0   Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
        NOARP  MTU:16  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:10
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)         Interrupt:18
 eth0   Link encap:Ethernet  HWaddr 00:50:c2:22:3b:0e
        UP BROADCAST MULTICAST  MTU:1500  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
 eth1   Link encap:Ethernet  HWaddr 00:50:c2:22:3b:60
        UP BROADCAST MULTICAST  MTU:1500  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
        Interrupt:41 Base address:0xe000   lo     Link encap:Local Loopback
 inet addr:127.0.0.1  Mask:255.0.0.0  inet6 addr: ::1/128 Scope:Host
 UP LOOPBACK RUNNING  MTU:16436  Metric:1
 RX packets:256 errors:0 dropped:0 overruns:0 frame:0
 TX packets:256 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:0   RX bytes:19952 (19.9 KB)  TX bytes:19952 (19.9 KB)
```

在上面的结果中，eth0 和 eth1 设备为以太网接口，can0 设备为 CAN 总线接口。接下来使用 ip 命令来配置 CAN 总线的位速率：

```
ip link set can0 type cantq 125 prop-seg 6phase-seg1 7 phase-seg2 2 sjw 1
```

也可以使用 ip 命令直接设定位速率：

```
ip link set can0 type can bitrate 125000
```

当设置完成后，可以通过下面的命令查询 can0 设备的参数设置：

```
ip -details link show can0
```

当设置完成后，可以使用下面的命令使能 can0 设备：

```
ifconfig can0 up
```

使用下面的命令取消 can0 设备使能：

```
ifconfig can0 down
```

在设备工作中，可以使用下面的命令来查询工作状态：

```
ip -details -statistics link show can0
```

7 CAN 总线应用开发接口

由于系统将 CAN 设备作为网络设备进行管理，因此在 CAN 总线应用开发方面，Linux 提供了 SocketCAN 接口，使得 CAN 总线通信近似于和以太网的通信，应用程序开发接口更加通用，也更加灵活。

此外，通过 https://gitorious.org/linux-can/can-utils 网站发布的基于 SocketCAN 的 can-utils 工具套件，也可以实现简易的 CAN 总线通信。

下面具体介绍使用 SocketCAN 实现通信时使用的应用程序开发接口。

(1) 初始化

SocketCAN 中大部分的数据结构和函数在头文件 linux/can.h 中进行了定义。CAN 总线套接字的创建采用标准的网络套接字操作来完成。网络套接字在头文件 sys/socket.h 中定义。套接字的初始化方法如下：

```
int s;

 struct sockaddr_can addr;

 struct ifreq ifr;

 s = socket(PF_CAN, SOCK_RAW, CAN_RAW);//创建 SocketCAN 套接字
 strcpy(ifr.ifr_name, "can0" );

ioctl(s, SIOCGIFINDEX, &ifr);//指定 can0 设备

addr.can_family = AF_CAN;

addr.can_ifindex = ifr.ifr_ifindex;
bind(s, (struct sockaddr *)&addr, sizeof(addr));    //将套接字与 can0 绑定
```

(2)数据发送

在数据收发的内容方面，CAN 总线与标准套接字通信稍有不同，每一次通信都采用 can_ frame 结构体将数据封装成帧。结构体定义如下：

```
struct can_frame {

canid_t  can_id;//CAN 标识符

__u8  can_dlc;//数据场的长度

__u8  data[8];//数据

};
```

can_id 为帧的标识符，如果发出的是标准帧，就使用 can_id 的低 11 位；如果为扩展帧，就使用 0～28 位。can_id 的第 29、30、31 位是帧的标志位，用来定义帧的类型，定义如下：

```
#define CAN_EFF_FLAG 0x80000000U    //扩展帧的标识
#define CAN_RTR_FLAG 0x40000000U    //远程帧的标识
#define CAN_ERR_FLAG 0x20000000U    //错误帧的标识，用于错误检查
```

数据发送使用 write 函数来实现。如果发送的数据帧(标识符为 0x123)包含单个字节(0xAB)的数据，可采用如下方法进行发送：

```
struct can_frame frame;

frame.can_id = 0x123;//如果为扩展帧，那么 frame.can_id = CAN_EFF_FLAG | 0x123;
frame.can_dlc = 1;     //数据长度为 1

frame.data[0] = 0xAB;     //数据内容为 0xAB
int nbytes = write(s, &frame, sizeof(frame)); //发送数据

if(nbytes != sizeof(frame))   //如果 nbytes 不等于帧长度，就说明发送失败
   printf("Error\n!");
```

如果要发送远程帧(标识符为 0x123)，可采用如下方法进行发送：

```
struct can_frame frame;

 frame.can_id = CAN_RTR_FLAG | 0x123;

write(s, &frame, sizeof(frame));
```

(3) 数据接收

数据接收使用 read 函数来完成，实现如下：

```
struct can_frame frame;

int nbytes = read(s, &frame, sizeof(frame));
```

当然，套接字数据收发时常用的 send、sendto、sendmsg 以及对应的 recv 函数也都可以用于 CAN 总线数据的收发。

4. 错误处理

当帧接收后，可以通过判断 can_id 中的 CAN_ERR_FLAG 位来判断接收的帧是否为错误帧。如果为错误帧，可以通过 can_id 的其他符号位来判断错误的具体原因。

错误帧的符号位在头文件 linux/can/error.h 中定义。

5. 过滤规则设置

在数据接收时，系统可以根据预先设置的过滤规则，实现对报文的过滤。过滤规则使用 can_filter 结构体来实现，定义如下：

```
struct can_filter {

canid_t can_id;

canid_t can_mask;};
```

过滤的规则为：

接收到的数据帧的 can_id &mask== can_id & mask

通过这条规则可以在系统中过滤掉所有不符合规则的报文，使得应用程序不需要对无关的报文进行处理。在 can_filter 结构的 can_id 中，符号位 CAN_INV_FILTER 在置位时可以实现 can_id 在执行过滤前的位反转。

用户可以为每个打开的套接字设置多条独立的过滤规则，使用方法如下：

```
struct can_filter rfilter[2];

 rfilter[0].can_id   = 0x123;
 rfilter[0].can_mask = CAN_SFF_MASK; //#define CAN_SFF_MASK 0x000007FFU
 rfilter[1].can_id   = 0x200;

 rfilter[1].can_mask = 0x700;
 setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));//设置规则
```

在极端情况下，如果应用程序不需要接收报文，可以禁用过滤规则。这样的话，原始套接字就会忽略所有接收到的报文。在这种仅仅发送数据的应用中，可以在内核中省略接收队列，以此减少 CPU 资源的消耗。禁用方法如下：

```
setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0); //禁用过滤规则
```

通过错误掩码可以实现对错误帧的过滤，例如：

```
can_err_mask_t err_mask = ( CAN_ERR_TX_TIMEOUT | CAN_ERR_BUSOFF );
setsockopt(s, SOL_CAN_RAW, CAN_RAW_ERR_FILTER, err_mask, sizeof(err_mask));
```

在默认情况下，本地回环功能是开启的，可以使用下面的方法关闭回环/开启功能：

```
int loopback = 0;  // 0表示关闭, 1表示开启(默认)
 setsockopt(s, SOL_CAN_RAW, CAN_RAW_LOOPBACK, &loopback, sizeof(loopback));
```

在本地回环功能开启的情况下，所有的发送帧都会被回环到与 CAN 总线接口对应的套接字上。默认情况下，发送 CAN 报文的套接字不想接收自己发送的报文，因此发送套接字上的回环功能是关闭的。可以在需要的时候改变这一默认行为：

```
int ro = 1; // 0表示关闭(默认), 1表示开启
 setsockopt(s, SOL_CAN_RAW, CAN_RAW_RECV_OWN_MSGS, &ro, sizeof(ro));

   /* 1.报文发送程序 */

   #include <stdio.h>

   #include <stdlib.h>

   #include <string.h>

   #include <unistd.h>
```

```
#include <net/if.h>

#include <sys/ioctl.h>

#include <sys/socket.h>

#include <linux/can.h>

#include <linux/can/raw.h>

int main()

{

    int s, nbytes;

    struct sockaddr_can addr;

    struct ifreq ifr;

    struct can_frame frame[2] = {{0}};

    s = socket(PF_CAN, SOCK_RAW, CAN_RAW);//创建套接字

    strcpy(ifr.ifr_name, "can0" );

    ioctl(s, SIOCGIFINDEX, &ifr); //指定 can0 设备

    addr.can_family = AF_CAN;

    addr.can_ifindex = ifr.ifr_ifindex;

    bind(s, (struct sockaddr *)&addr, sizeof(addr));//将套接字与 can0 绑定

    //禁用过滤规则，本进程不接收报文，只负责发送

    setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, NULL, 0);

    //生成两个报文

    frame[0].can_id = 0x11;

    frame[0]. can_dlc = 1;

    frame[0].data[0] = 'Y';

    frame[0].can_id = 0x22;

    frame[0]. can_dlc = 1;

    frame[0].data[0] = 'N';
```

```
    //循环发送两个报文
    while(1)
    {
        nbytes = write(s, &frame[0], sizeof(frame[0]));     //发送 frame[0]
        if(nbytes != sizeof(frame[0]))
        {
            printf("Send Error frame[0]\n!");
            break; //发送错误，退出
        }
        sleep(1);
        nbytes = write(s, &frame[1], sizeof(frame[1]));     //发送 frame[1]
        if(nbytes != sizeof(frame[0]))
        {
            printf("Send Error frame[1]\n!");
            break;
        }
        sleep(1);
    }
    close(s);
    return 0;
}
/* 2. 报文过滤接收程序 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
#include <net/if.h>

#include <sys/ioctl.h>

#include <sys/socket.h>

#include <linux/can.h>

#include <linux/can/raw.h>

int main()

{
    int s, nbytes;
    struct sockaddr_can addr;
    struct ifreq ifr;
    struct can_frame frame;
    struct can_filter rfilter[1];
    s = socket(PF_CAN, SOCK_RAW, CAN_RAW); //创建套接字
    strcpy(ifr.ifr_name, "can0" );
    ioctl(s, SIOCGIFINDEX, &ifr); //指定 can0 设备
    addr.can_family = AF_CAN;
    addr.can_ifindex = ifr.ifr_ifindex;
    bind(s, (struct sockaddr *)&addr, sizeof(addr)); //将套接字与 can0 绑定
    //定义接收规则，只接收表示符等于 0x11 的报文
    rfilter[0].can_id   = 0x11;
    rfilter[0].can_mask = CAN_SFF_MASK;
    //设置过滤规则
    setsockopt(s, SOL_CAN_RAW, CAN_RAW_FILTER, &rfilter, sizeof(rfilter));
     while(1)
     {
       nbytes = read(s, &frame, sizeof(frame));  //接收报文
```

```
        //显示报文
        if(nbytes > 0)
        {
            printf(“ID=0x%X DLC=%d data[0]=0x%X\n”, frame.can_id,
            frame.can_dlc, frame.data[0]);
        }
    }
    close(s);
    return 0;
}
```